



# Angular

Formation de base

Expliqué par David Gilson

<https://github.com/gilsdav>

# Outils

- ▶ NodeJS (<https://nodejs.org>) LTS+
- ▶ Visual Studio Code (<https://code.visualstudio.com>)
  - ▶ Extensions :
    - ▶ Angular Extension Pack
    - ▶ Angular Files
- ▶ Angular Cli ((sudo) npm install -g @angular/cli)
- ▶ Chrome (<https://www.google.com/intl/fr/chrome/browser>)
  - ▶ Extension :
    - ▶ Augury : <https://chrome.google.com/webstore/detail/augury/elgalmkoelokbchhkhacckoklkejnhcd>

# Sommaire

- ▶ Pourquoi du Javascript ?
- ▶ Présentation de NodeJS
- ▶ Présentation TypeScript
- ▶ Présentation Angular
  - ▶ Pourquoi passer à Angular ?
  - ▶ TypeScript ou JavaScript
  - ▶ Components
  - ▶ Directives
  - ▶ Services
  - ▶ Pipes
  - ▶ Data Binding
  - ▶ Routing
  - ▶ Guards
  - ▶ Forms
  - ▶ Guidelines / Organisation du code

# Pourquoi du JavaScript ?

- ▶ Complètement indépendant de toute plateforme
  - ▶ Browser
  - ▶ Server
    - ▶ IBM LoopBack
    - ▶ Sails
    - ▶ ...
  - ▶ Desktop
    - ▶ Electron
  - ▶ Mobile
    - ▶ Cordova
    - ▶ NativeScript
  - ▶ IOT
    - ▶ Intel XDK
    - ▶ Johnny-five
    - ▶ Cylon
    - ▶ ...

# Mais encore

- ▶ SPA (Single Page Application)
  - ▶ Meilleur expérience utilisateur car plus fluide
- ▶ PWA
  - ▶ VS Site web
    - ▶ Accès hors ligne
    - ▶ Accès à certaines fonctionnalités natives (<https://whatwebcando.today>)
  - ▶ VS Application mobile
    - ▶ Zero-install
    - ▶ Pas besoin de store (mais vont bientôt apparaître dans le store de Microsoft)
    - ▶ Mise à jour à l'utilisation
    - ▶ Pas accès à tous les capteurs
  - ▶ Le Offline ne fonctionne pas encore sur iOS

# 1) NodeJS

# NodeJS ?

- ▶ NodeJS est une technologie libre qui offre la possibilité d'exécuter du JavaScript en dehors d'un navigateur. Il est donc utilisé pour :
  - ▶ Crée des applications serveur.
  - ▶ Créer des scripts de build/test/...
- ▶ NodeJS intègre un puissant gestionnaire de package (npm).
- ▶ Basé sur le moteur JS V8, et ayant vu le jour en 2009, il est jeune, mais a déjà beaucoup d'expériences.
- ▶ NodeJs ne nécessite pas de serveur http comme Apache, Tomcat ou IIS, il contient une bibliothèque HTTP (express).

## 2) TypeScript

# TypeScript ?

- ▶ TypeScript est un langage de programmation libre et open-source développé par Microsoft qui a pour but de simplifier la création d'applications web.
- ▶ C'est un sur-ensemble de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript).
- ▶ Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.
- ▶ TypeScript permet un typage statique optionnel des variables et des fonctions, la création de classes et d'interfaces, l'import de modules, tout en conservant l'approche non-constrictrice de JavaScript.

# TypeScript - exemples

## Typage Statique

```
// Création d'une variable contenant une valeur booléenne.  
const aValeurBooleenne: boolean = false;  
  
// Création d'une variable contenant une chaîne de caractère.  
const maChaineDeCaractere string = "Hello World";  
  
// Création d'une variable contenant un nombre.  
const monNombre number = 1;  
  
// Création d'une fonction retournant une chaîne de caractère.  
  
function maFonction(): string {  
    return "Ma valeur de retour";  
}
```

## Typage Générique

```
function maFonction<T>(parametre: T) {  
    // Contenu de la fonction.  
}
```

```
class MaClasse<T> { maVariable : T; // Contenu de la classe. }  
// Création d'une instance de la classe "MaClasse" en définissant un type.  
const maClasse= new MaClasse<string>();  
maClasse.maVariable = "Hello World";
```

# TypeScript - class modifiers

- ▶ **public** : accès autorisé depuis l'extérieur de la classe (nécessaire pour les propriétés utilisées dans l'HTML)
  - ▶ L'utilisation de public sur un attribut une méthode/délégué dans une classe équivaut à ne rien mettre
  - ▶ Mettre public ou private sur un paramètre du constructeur de la classe rend ce paramètre accessible depuis la classe entière alors que ne rien mettre encapsule ce paramètre dans le constructeur.
- ▶ **private** : accès uniquement autorisé à l'intérieur de la class
- ▶ **protected** : accès autorisé depuis la classe et ses enfants (héritage)

# TypeScript - variable declarations

- ▶ **var** : à ne pas utiliser ! Sauf quand vous devez accéder à une variable existante en JS (créée par une librairie par exemple). Il s'agit d'une déclaration JavaScript.
  - ▶ declare var varFromJs: any;
- ▶ **const** : à utiliser quand vous ne savez pas si vous allez modifier la référence de la variable. (déclaration à utiliser par défaut)
- ▶ **let** : à utiliser si vous modifiez la référence de la variable après son initialisation

# TypeScript - plus d'informations

- ▶ Site officiel :
  - ▶ <https://www.typescriptlang.org>

# Immutabilité

- ▶ Mutation
  - ▶ Presque tout en JavaScript est mutable. Cela veut dire que l'on peut modifier un objet sans modifier sa référence.
  - ▶ La mutation est très rapide
  - ▶ Exemple d'éléments mutables: Function, Object, Array
- ▶ Immutabilité
  - ▶ Changer la référence permet de pouvoir savoir plus rapidement que quelque chose a changé dans un objet (pas besoin de deepChecking). Meilleur pour les Input Angular.
  - ▶ La manipulation d'objets de façon immutable est plus lente
  - ▶ Exemple d'éléments immutables: Number, String

# Immutabilité

- ▶ Exemple modification par mutation

```
const testValue = {test: 'test'}; // ref = 1
testValue.test = 'test2';
// resultat: {test: 'test2'} => ref = 1
```

- ▶ Exemple modification immutable

```
const testValue = {test: 'test'}; // ref = 1
testValue = {...testValue, test: 'test2'};
// resultat: {test: 'test2'} => ref = 2
```

- ▶ Attention le Spread Operator (...) ne fait pas de copie profonde (deep copy)

# Lambda vs Function

- ▶ **Function** : `this` fait référence à l'objet qui a fait **appel** à la fonction

```
function() {}
```

- ▶ **Lambda** (fonction fléchée) : `this` fait référence à l'objet où elle est **créée**

```
() => {}
```

### 3) Présentation Angular (2+)

# Pourquoi passer à Angular 2+ ?

- ▶ Le but d'Angular est de faire une application multiplateforme et réutilisable permettant de faire des applications Web, Mobile et Desktop.
- ▶ *Il est plus rapide qu'AngularJs*
- ▶ *La création de composants est plus simple que l'association contrôleur - scope - directive d'AngularJs*
- ▶ Syntaxe dite « plus intuitive »
- ▶ Améliorations au niveau des services

# TypeScript ou Javascript ?

- ▶ TypeScript permet la transpilation (transcompilation) vers du JavaScript compris par des navigateurs assez vieux si nécessaire
- ▶ Le core d'Angular est programmé en TypeScript
- ▶ Le typage permet un coding plus rapide et permet moins d'erreurs

# RxJS - reactive programming

- ▶ Observable: Stream dans lequel on peut envoyer des données ou les recevoir.
  - ▶ Observer: Objet qui gère le Stream. On peut lui demander d'envoyer une nouvelle valeur, une erreur ou de clôturer le Stream.
  - ▶ Subscription: Objet qui écoute le Stream en attendant une valeur pour exécuter une action.

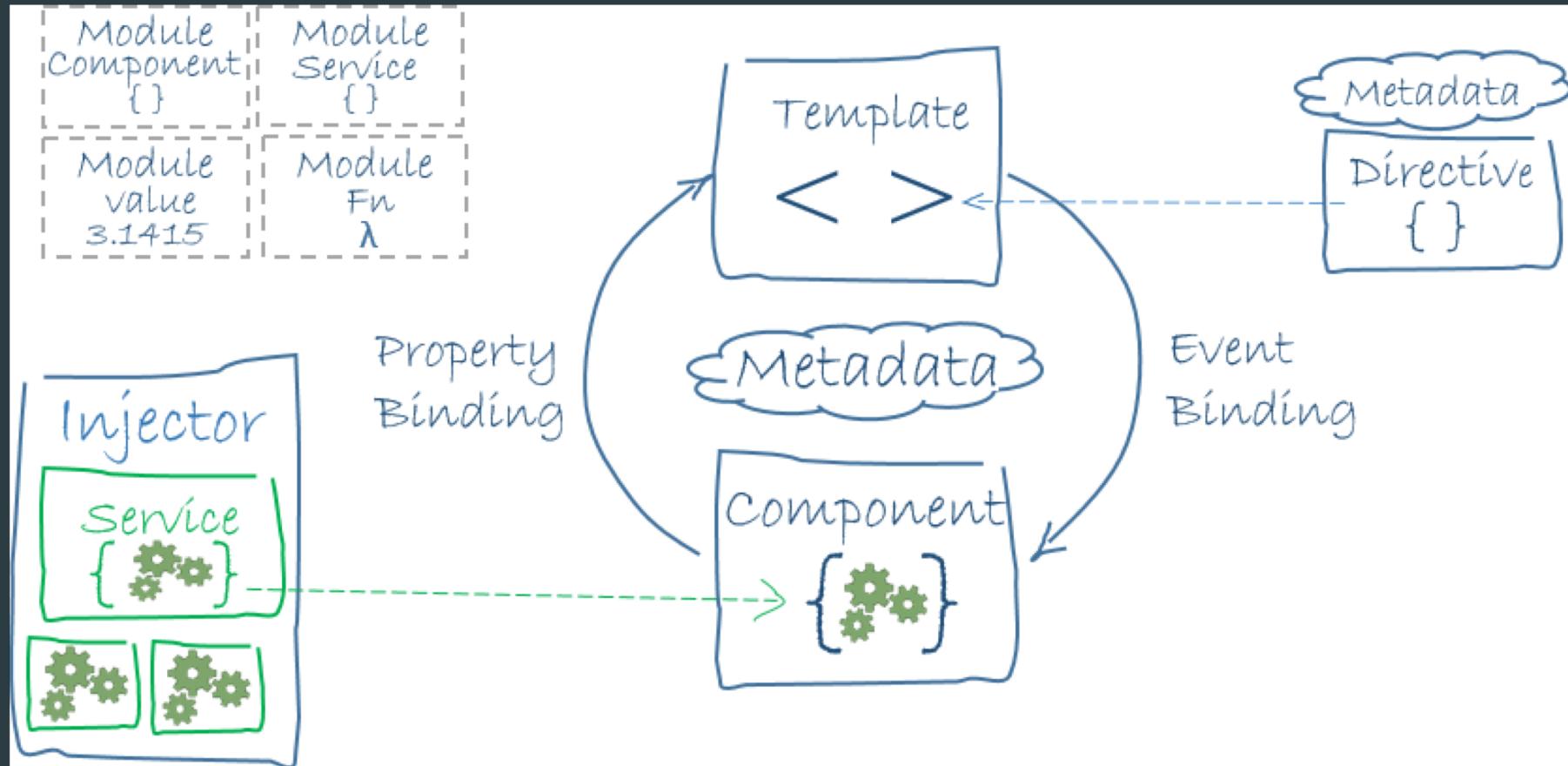
```
const sayHello: Observable<string> = Observable.create((observer: Observer<string>) => {  
  setTimeout(() => {  
    observer.next('hello');  
    observer.complete();  
  }, 10000);  
});  
  
sayHello.subscribe(value => {  
  console.log(value); // hello  
});
```

Plus d'informations : <https://codecraft.tv/courses/angular/reactive-programming-with-rxjs/observables-and-rxjs/>

# Concept de base

- ▶ Le principe de base est la découpe des écrans en vues partiels par fonctionnalités ou pour les vues utilisées à plusieurs endroits : les *components*
- ▶ Angular génère le dom ou les parties de dom nécessaires à l'application.
  - ▶ *Dans AngularJS, angular s'occupai de parser le dom pour mettre à jour les valeurs et mettre en place le binding.*

# Concept de base



### 3) Les modules

# Modules

- ▶ Angular est modulaire. Il propose de son propre système de modules "NgModule".
- ▶ Les modules autres que l'AppModule, SharedModule ou les libraires sont appelés des Feature Modules
- ▶ Un module peut contenir
  - ▶ Components
  - ▶ Services
  - ▶ Pipes
  - ▶ Guards
  - ▶ D'autres modules

# Modules

- ▶ Pourquoi faire plusieurs modules ?
  - ▶ Réutilisabilité interne à l'application
  - ▶ Réutilisabilité entre applications (librairies)
  - ▶ Performance (LazyLoad)

# Modules

- ▶ Exemple d'un module vide

```
@NgModule({  
  providers: [],  
  declarations: [],  
  entryComponents: []  
  imports: [],  
  exports: [],  
  bootstrap: []  
})  
class AppModule { }
```

# Modules

- ▶ **providers** : Un provider est la façon d'instancier un injectable (service/guard).  
Lister les services et les guards ici pour qu'ils deviennent injectables (utilisables).
- ▶ **declarations** : Déclarez ici les différents components/directives/pipes afin de pouvoir utiliser leurs selector/name.
- ▶ **entryComponents** : Permet de déclarer au transpileur que nous avons besoin de components qu'il pense non utilisés. Cela arrive quand on fait une instantiation dynamique d'un component. (Il est également nécessaire de le déclarer)
- ▶ **import** : Liste des modules à importer (aussi bien internes que les librairies).
- ▶ **exports** : Liste des components/pipes à mettre à disposition d'autres modules.
- ▶ **bootstrap** : Components indépendants.

# Modules - built-in

- ▶ **@angular/core** : la base d'Angular (requis)
- ▶ **@angular/common** : directives et services commun (requis)
  - ▶ **@angular/common/http** : services permettant des appels http
- ▶ **@angular/compiler** : librairie de compilation (JIT/AOT)
  - ▶ **@angular/compiler-cli** : CLI for @angular/compiler
- ▶ **@angular/platform-browser** : permet l'utilisation d'Angular dans un navigateur
  - ▶ Il existe aussi **@angular/platform-server** pour le faire tourner sur NodeJS
- ▶ **@angular/router** : gestion des routes
- ▶ **@angular/forms** : directives et services pour la création de formulaires

## 4) Les components

# Components

- ▶ Créer un *component*, c'est créer un nouveau tag HTML
- ▶ Dans une application Angular, une application est entièrement composée de *component* qui s'imbrique / se réutilise au besoin.
  - ▶ Une page est un component
  - ▶ Une partie de page est un component
  - ▶ ...
- ▶ Un component est une classe décorée regroupant du code TypeScript, HTML (et CSS/SCSS). Un système de binding (one way ou two ways) permet la communication entre le TypeScript et l'HTML.
  - ▶ L'HTML et le CSS peuvent être écrits en "inline" dans le décorateur ou bien séparés dans des fichiers.
    - ▶ template: `` vs templateUrl: "
    - ▶ styles: [] vs styleUrls: []
- ▶ Un component est identifié à l'aide d'un selector. Celui-ci représente le tag à utiliser dans l'HTML pour utiliser ce component.
- ▶ *Dans la version 1 d'angular, l'équivalent le plus proche des component est la directive.*

# Exemple (Master/Detail)

- Luke Skywalker
- Anakin Skywalker
- Obi-Wan Kenobi
- Olivier Lallemand
- Han Solo
- Leia Organa

Un *component* liste qui contient une liste

Un *component* de détail pour l'élément de la liste sélectionner

# Le component principal

- ▶ L'application est également un *component*
- ▶ Un *component* est composé de deux parties : une classe et son décorateur

```
@Component({  
  selector: 'my-app',  
  template: '<h1>My App</h1>',  
  providers: []  
}  
class AppComponent {}  
  
bootstrap(AppComponent)
```

```
<body>  
  <my-app>Loading...</my-app>  
</body>
```

- ▶ Le bootstrapping sert à instancier chaque composant du tableau pour les injecter dans le dom, en l'occurrence, ici c'est le composant de base de l'application qui sera injecté

## 5) Directives

# Directives structurelles

- ▶ Une directive structurelle modifie le Layout en ajoutant ou supprimant des éléments du DOM.
- ▶ Les trois principales sont : nglf, ngSwitch et ngFor

```
<div *ngIf="name">{{name}}</div>

<div *ngFor="let element of elements">{{element}}</div>

<div [ngSwitch]="status">
  <template [ngSwitchCase]=""ok">OK</template>
  <template [ngSwitchCase]=""ko">KO</template>
  <template ngSwitchDefault>NO</template>
</div>
```

# Directives structurelles

- ▶ Il est possible de créer sa propre directive structurelle
- ▶ Mettre l'étoile devant le nom de la directive nous permet d'avoir accès au ViewContainerRef et au TemplateRef
  - ▶ ViewContainerRef est une référence vers un endroit du DOM où l'on peut ajouter un component ou un template.
    - ▶ Il est donc possible de vider le ViewContainerRef: clear()
    - ▶ Et ajouter un template dans le ViewContainerRef: createEmbeddedView(this.templateRef)
  - ▶ TemplateRef est le template sur lequel est placée la directive.

```
this.viewContainerRef.createEmbeddedView(this.templateRef);
```

```
this.viewContainerRef.clear();
```

# Directives non structurelles

- ▶ Une directive non structurelle permet d'ajouter un comportement à un composant/tag HTML existant.
- ▶ Il s'utilise en tant qu'attribut.
- ▶ Pourquoi faire cela ? Créer un type de validation d'input qui n'existe pas de base (exemple: numéro de plaque).

```
@Directive({  
  selector: '[validPlateNumber]',  
  providers: [  
    { provide: NG_VALIDATORS, useExisting: PlateNumberValidator, multi: true }  
  ]  
})  
  
export class PlateNumberValidator implements Validator {}
```

## 6) Services

# Service

- ▶ Morceau de logique partageable (injectée) dans l'application.
- ▶ Quand utiliser les services ?
  - ▶ Pour toutes logiques non visuelle comme un appel service REST, un calcul...
- ▶ Injection possible à plusieurs endroits
  - ▶ Dans le module: Assure que le service soit singleton (une instance pour l'application)
  - ▶ Dans le component: Assure de créer une instance spécifique pour le component en question et ces enfants

```
@Injectable()
```

```
export class UserService {
```

## 7) Pipes



# Pipes

- ▶ Un pipe est une couche d'interprétation/manipulation de données
- ▶ Il est principalement utilisé depuis l'HTML mais peut également être utilisé en TypeScript.
- ▶ Il a plusieurs utilités:
  - ▶ Permettre de déterminer comment afficher un objet
  - ▶ Filtrer les données d'une boucle (ngFor)
- ▶ Il existe déjà certains Pipes comme: Date, UperCase, LowerCase, Currency et Percent
- ▶ *C'est l'équivalent des filtres Angular 1*

```
<p>The hero's birthday is {{ birthday | date:" dd/MM/yy" }}</p>
```

# Pipes

- ▶ Utilisation en tant que filtre

```
<ng-container *ngFor="let item of (list | myFilter:valueToFilter)">  
</ng-container>
```

- ▶ Création d'un Pipe

```
@Pipe({  
  name: 'myFilter'  
)  
  
export class MyFilterPipe implements PipeTransform {  
  transform(value: any, args?: any): any {  
  }  
}
```

# Pipes - pure vs impure

- ▶ **Pure** (par défaut) : n'est réexécuté que si la référence de la valeur ou des arguments a changée. Cela permet un gain de performance non négligeable.
- ▶ **Impure** : le pipe sera réexécuté à chaque possibilité de changement (interaction de l'utilisateur avec une partie du dom ou événement d'un observable reçu...).
  - ▶ Cela peut être nécessaire si le pipe a besoin d'une valeur externe non passée par argument (exemple: langue de l'application).

```
@Pipe({  
  name: 'myFilter',  
  pure: false  
})  
  
export class MyFilterPipe implements PipeTransform {  
  constructor(private service: myService) {}  
  transform(value: any, args?: any): any {}  
}
```

## 8) Data Binding

# Data Binding

- ▶ Côté parent
  - ▶ Les [] représentent un data binding unidirectionnel model (TS) -> vue (HTML)
  - ▶ Les () représentent un data binding unidirectionnel vue (HTML) -> model (TS)
  - ▶ La combinaison [()] représente donc un data binding bidirectionnel

```
<my-comp [data]="data"></my-comp>
```

```
<my-comp (selection)="selection = $event"></my-comp>
```

```
<my-comp (click)="onClick()"></my-comp>
```

```
<input type="text" [(ngModel)]="model.name" name="name" />
```

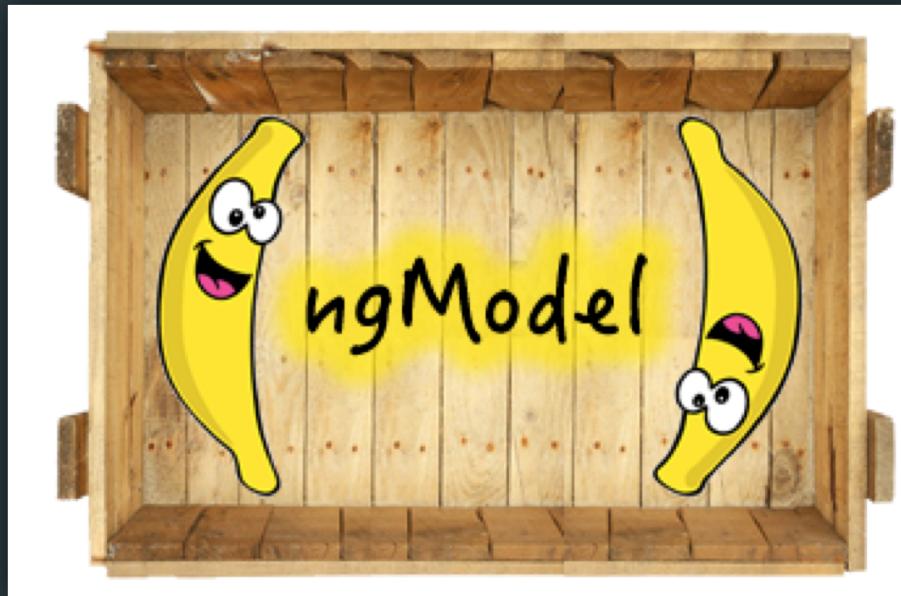
```
<input type="text" name="name"  
[ngModel]="model.name"  
(ngModelChange)="model.name = $event" />
```

# Data Binding

- ▶ Côté enfant
  - ▶ [] = @Input()
  - ▶ () = @Output()
    - ▶ Il s'agit un EventEmiter
  - ▶ [()] = @Input() + @Output()
    - ▶ L'output doit avoir le même nom que l'input suivit de "change"
      - ▶ Input: data
      - ▶ Output: dataChange
  - ▶ [(ngModel)] = NG\_VALUE\_ACCESSOR
    - ▶ ngModel est réservé aux accessor
      - ▶ Un accessor est soit utilisable via
        - ▶ ngModel pour des formulaires en template driven (logique dans l'HTML)
        - ▶ Value pour des formulaires en reactive form (logique dans le TS)

# Data Binding

- ▶ Moyen mémo technique pour le two ways



## 9) Routing

# Routes

- ▶ La gestion d'accès aux différentes pages se fait via l'URL
- ▶ Pour savoir quelle page afficher avec quelle URL, il faut mettre en place le routing.

```
export const routes: Routes = [
  { path: "", component: ParentComponent, children: [
    { path: "", redirectTo: '/home', pathMatch: 'full' },
    { path: 'home', loadChildren: './home/home.module#HomeModule',
      canActivate: [AuthGuard] },
    { path: 'login', loadChildren: './login/login.module#LoginModule' },
  ] },
  { path: '**', redirectTo: '/home' }
];
```

```
RouterModule.forRoot(routes)
```

```
<router-outlet></router-outlet>
```

# 10) Guards

# Guards

- ▶ Un Guard permet de gérer l'accès aux pages (sécuriser la navigation) internes à l'application
- ▶ Deux types de Guards:
  - ▶ OnActivate: Avant d'afficher la page
  - ▶ OnDeactivate: Avant de quitter la page

```
@Injectable()  
export class AuthGuard implements CanActivate {
```

```
@Injectable()  
export class AuthGuard implements CanDeactivate {
```

# 11) Forms

# Forms

- ▶ Il existe deux façons de faire des formulaires :
  - ▶ **Template driven**
    - ▶ Définition du binding et des validations dans l'HTML
  - ▶ **Reactive form**
    - ▶ Définition du binding et des validations dans le TS

# Template driven form

- ▶ Gestion du binding des contrôles input à l'aide de [(ngModel)]
- ▶ Définition de validation à l'aide de directives
- ▶ Accès aux données du formulaire (statuts de validation) à l'aide d'une variable de template
- ▶ Mutation

```
<form #myForm="ngForm">
  <input type="text" id="myInput" name="myInput" #myInput="ngModel"
    required [(ngModel)]="myModel.myVariable"/>
  <span *ngIf="!myInput.valid && myInput.dirty">Requis !</span>
</form>

console.log(this.myModel.myVariable);
```

# Reactive form

- ▶ Modèle dédié à l'HTML + Modèle business
- ▶ Création et manipulation d'un arbre de contrôles en TS qui comprend
  - ▶ La valeur de base du model dédié à l'HTML
  - ▶ La validation des contrôles
- ▶ Immutabilité

```
this.myForm = this.fb.group({ myInput: [this.myVariable, Validators.required ] });
```

```
<form [formGroup]="myForm">
  <input id="myInput" name="myInput" formControlName="myInput"/>
</form>
```

```
console.log(this.myForm.get('myInput').value);
```

# Form component

- ▶ Outre l'utilisation des inputs/select/textarea standards, il est possible de créer votre component de formulaire qui permet l'utilisation du **[(ngModel)]** ou encore d'un **formControlName**.
- ▶ Pour ce faire il faut utiliser
  - ▶ **NG\_VALUE\_ACCESSOR** : gestion de la valeur du control
  - ▶ **NG\_VALIDATORS** : validation custom

# Form component - value accessor

```
@Component({  
  ...  
  providers: [  
    {  
      provide: NG_VALUE_ACCESSOR,  
      useExisting: forwardRef(() => MyComponent),  
      multi: true  
    }  
  ]  
})  
export class MyComponent implements ControlValueAccessor {}
```

# Form component - value accessor

- ▶ Fonctions à implémenter :
  - ▶ `writeValue(...)` : récupérer la valeur depuis le formulaire
  - ▶ `registerOnChange(...)` : enregistrer le callback à appeler quand il y a un changement de valeur à transmettre au formulaire
  - ▶ `registerOnTouched(...)` : enregistrer le callback à appeler pour informer le formulaire que votre component a été touché
  - ▶ `setDisabledState(...)` : récupérer l'état d'activabilité depuis le formulaire

# Form component - validator

```
@Component({  
  ...  
  providers: [  
    {  
      provide: NG_VALIDATORS,  
      useExisting: forwardRef(() => MyComponent),  
      multi: true  
    }  
  ]  
})  
export class MyComponent implements Validator {}
```

# Form component - validator

- ▶ Fonctions à implémenter :
  - ▶ validate(...) : renvoie une erreur s'il y en a une
    - ▶ Pas d'erreur : return null;
    - ▶ Cas d'erreur : return { customErrorName: true };

## 12) Guidelines

<https://angular.io/guide/styleguide>

# Petit mot de sécurité

- ▶ Faites très attention aux librairies que vous utilisez.
  - ▶ Utilisez des librairies scopées (@scope) exemple: @angular/ ...
    - ▶ Si vous avez confiance en la société /personne derrière le scope, vous pouvez normalement avoir confiance en toutes ces librairies
  - ▶ Utilisez des librairies populaires (nombre de téléchargements élevé)
  - ▶ N'hésitez pas à mettre votre nez dans le code de la librairie
  - ▶ Mettez à jour régulièrement le Framework et les librairies
- ▶ Une faute de frappe comme « boostrap » à la place de « bootstrap » peut avoir des effets désastreux pour la vie privée des clients.

# Liens utiles

- ▶ <https://angular.io>
- ▶ <https://github.com/angular/angular-cli/wiki>
- ▶ <https://www.npmjs.com>
- ▶ <https://material.angular.io>
- ▶ <http://www.ngx-translate.com>
- ▶ <https://www.learnrxjs.io>
  
- ▶ <https://getbootstrap.com>
- ▶ <https://valor-software.com/ngx-bootstrap>
  
- ▶ <https://lodash.com>

# Références

- ▶ <https://angular.io>
- ▶ <https://toddmotto.com/angular/>
- ▶ <https://codecraft.tv/courses/angular/>
- ▶ <https://wetry.io>

# Autres sujets de formation

- ▶ Angular avancé
  - ▶ Angular Universal (SSR/SEO)
  - ▶ NGRX (Angular avec Redux)
  - ▶ Progressive Web App
- ▶ Autres
  - ▶ Lodash